Tuomas Hyttinen
1503589
UNIX Operating systems
Spring 2017

The database

I started off with acquiring a server instance from Amazon Web Services as I needed a platform for my database and didn't want to use the pre-installed one provided by Metropolia.

I was a little hasty when putting up the server and didn't notice the instance was physically located in Portland, Oregon, which apparently creates somewhat of an overhead because of prolonged data fetching time.

The database is running an Ubuntu Server 16.04 (Xenial Xerus) distribution. I chose a Debian based distro mostly because I was more familiar with the APT package handler than RPM for example.

After installing MySQL I created the database shop_db with two tables: users and products.

MySQL installation by default defines the root database user that has unlimited permissions to operate on the database server. It is not good practice, though, to use the root user to gain remote access in a web application, so I created a new user just for that. For security reasons I only defined the user for the users.metropolia.fi hostname. Connections attempted from any other address will fail.

> ***CREATE USER 'superuser'@'195.148.105.101' IDENTIFIED BY [password];***

Next, I granted the user only the privileges needed for the system to work:

> ***GRANT SELECT, INSERT, UDPATE, DELETE on shop_db.products to 'superuser'@'195.148.105.101';***
>
> ***GRANT SELECT on shop_db.users to 'superuser'@'195.148.105.101';***

In addition I had to comment out the line

```
        bind-address = 127.0.0.1
```

from MySQL configuration file and open port 3306 in iptables for inbound traffic to allow remote access.

About the shop_db.users table

The table stores three different values on each user: username, password hash and user role.

Username and password hash are stored as strings (VARCHAR) and the user role is an integer value (1 for customer, 2 for store employee, 3 for admin user)

Tuomas Hyttinen
1503589
UNIX Operating systems
Spring 2017

The hash is produced by **password_hash** function integrated to PHP. It uses the ***bcrypt*** algorithm, that is sufficiently strong for storing sensitive data. The pre-built function is quite handy, because it includes the password salt in the hash itself. This way there is no need to store the salt separately along the user information. The hash generated in this manner can also easily be compared to a user-provided plaintext password using ***password_verify*** .

The web store application

When entering the web store, the user first sees a simple login page with username and password fields and a submit button. The form triggers a http POST request to a server side script ***authenticate.php***, passing the username and password to the script as parameters.

First, a separate PHP script **connect.php** is included in the file. It establishes a database connection as a MYSQLI object that is globally available across the session.

The authentication script then invokes a function called ***authenticate_user*** that takes the provided username and password, fetches the user from the database, hashes the plaintext password, compares the produced hash to the one found in the DB and, on a match, returns TRUE and otherwise FALSE.

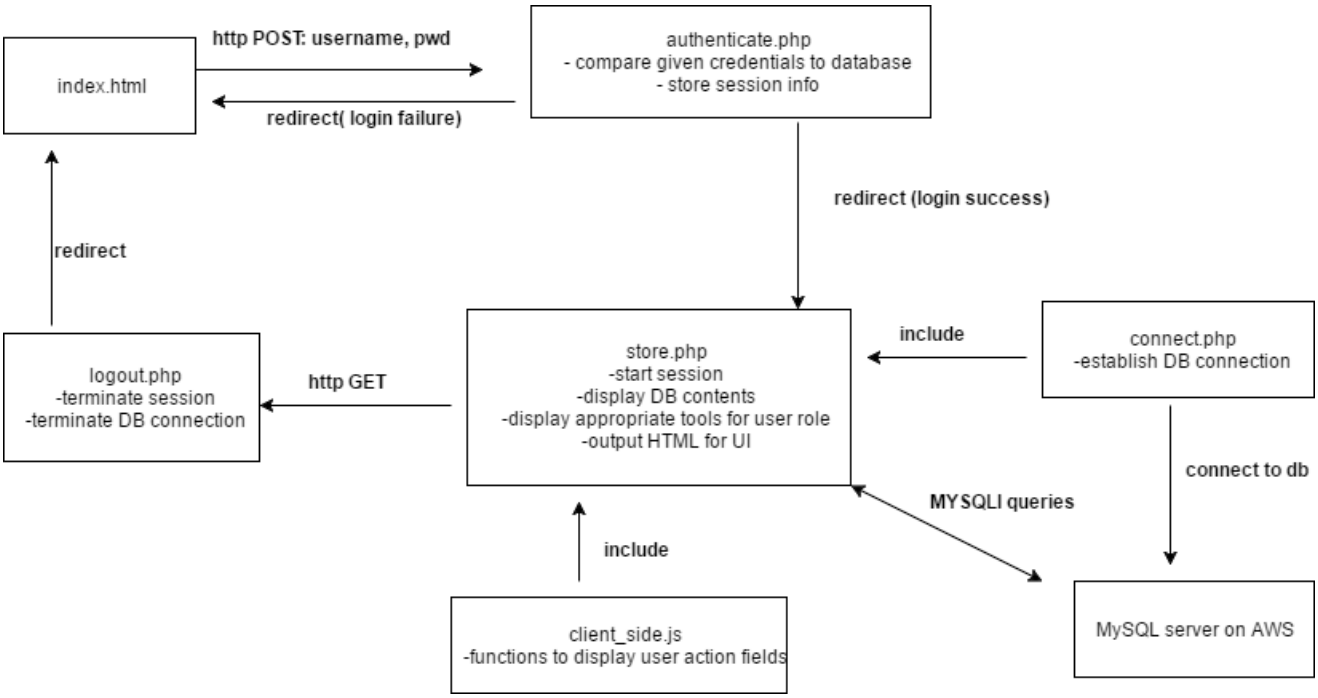

Figure 1: flow diagram for the application

Tuomas Hyttinen
1503589
UNIX Operating systems
Spring 2017

After a successful authentication the script stores the username and user role to superglobal $_SESSION variables that are accessible throughout the session. It then sends out a raw http header to the user containing the *location* parameter that redirects the user to the main application ***store.php*.**

It defines cookie parameters for the session (I set the cookies to expire after 1 hour of idle time and to only work within the ~/tuomahy subdomain) and initiates a new session. It then displays the contents of the database with ***display_products*** function.

The user input fields for different database operations (buy, increment number of products, add new products and remove product) are displayed by a separate function ***display_db_tools*** that outputs the correct HTML elements to the browser according to the ***user_role*** session variable that was set earlier.

A little bit of JavaScript is also included in a separate client_side.js file to toggle the visibility of the input fields according to the action that the user chooses (for instance when you are buying a product, you don't need to see the field for inserting a name for a brand new product).

The functions for the actual database queries are also included in **store.php** : ***buy_products***, ***increment***, ***remove_product*** and ***add_new_product***

Finally, when the user wants to end the session, the logout button will trigger a GET request to a the script ***logout.php*** that destroys the current session, terminates the database connection and redirects the user to the login page.

Security aspects

Concerning data security, I tried to do everything as by-the-book as I could. There probably are still some loopholes in the construct as I'm not a seasoned web developer, but nothing too severe, I hope.

Every single database query in this application is conducted using prepared statements to prevent SQL injections.

The user input from the login page is passed through functions that strip slashes and HTML-style tags from the text to prevent Cross Site Scripting.

The information needed to establish the database connection (host, password etc.) are defined as constants in a separate configuration file ***dbcnn_conf.php*** stored away from the server's document root and chmod for the directory set to 751 to prevent unauthorized access.

For a safe login, the website should be accessed through a secure HTTPS connection, since the application itself does not encrypt user credentials as they are sent over to the server. However, plaintext passwords are never stored anywhere, instead a password hashing mechanism with salting is used, as described earlier.